注: 1.阅读此篇前你需要先阅读我的之前三篇文章:XmlBeanFactory源码分析(上),XmlBeanFactory源码分析(下),FileSystemXmlApplicationContext源码分析

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-aspects</artifactId>
    <version>4.3.12.RELEASE</version>
</dependency>
```

解析入口的测试代码: 主类:

```
public class MathCalculator {

    public float div(int i,int j) {

        System.out.println("MathCalculator .... div() ...");
        return i/j;
    }

}
```

切面类:

```
//@Aspect注解表明这是一个切面类
@Aspect
public class LogAspect {

    //抽取公共的切入点表达式
    //1、本类引用
    //2、其他的切面引用
    @Pointcut("execution(public float com.mgw.aop.MathCalculator.*(..))")
    public void pointCut(){};

    //@Before在目标方法之前切入；切入点表达式（指定在哪个方法切入）
    @Before("pointCut()")
    public void logStart(JoinPoint joinPoint){
        Object[] args = joinPoint.getArgs();
        System.out.println(""+joinPoint.getSignature().getName()+"运行。。。@Before:参数列表是：{"+
Arrays.asList(args)+"}");
    }

    @After("com.mgw.aop.LogAspect.pointCut()")
    public void logEnd(JoinPoint joinPoint){
        System.out.println(""+joinPoint.getSignature().getName()+"结束。。。@After");
    }

    //JoinPoint一定要出现在参数表的第一位
    @AfterReturning(value="pointCut()",returning="result")
    public void logReturn(JoinPoint joinPoint,Object result){
        System.out.println(""+joinPoint.getSignature().getName()+"正常返回。。。@AfterReturning:运行结
果：{"+result+"}");
    }
```

```
    @AfterThrowing(value="pointCut()",throwing="exception")
    public void logException(JoinPoint joinPoint,Exception exception){
        System.out.println(""+joinPoint.getSignature().getName()+"异常。。。异常信息:
{"+exception+"}");
    }
}
```

配置类:

```
//@EnableAspectJAutoProxy注解表明开启切面编程
@EnableAspectJAutoProxy
@Configuration
public class MainConfigOfAop {

    @Bean
    public MathCalculator mathCalculator() {

        return new MathCalculator();
    }

    @Bean
    public LogAspect logAspect() {

        return new LogAspect();
    }

}
```

测试函数:

```
@Test
public void test01() {

    AnnotationConfigApplicationContext applicationContext =
            new AnnotationConfigApplicationContext(MainConfigOfAop.class);

    MathCalculator mathCalculator = applicationContext.getBean(MathCalculator.class);

    //mathCalculator.div(2,1);
    mathCalculator.div(2,0);
}
```

解析开始: @EnableAspectJAutoProxy这个注解是整个aop的入口

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Import(AspectJAutoProxyRegistrar.class)
public @interface EnableAspectJAutoProxy {

    boolean proxyTargetClass() default false;

    boolean exposeProxy() default false;
```

```
}
//@Import(AspectJAutoProxyRegistrar.class)为IOC容器注入AspectJAutoProxyRegistrar这个类
class AspectJAutoProxyRegistrar implements ImportBeanDefinitionRegistrar {
    @Override
    public void registerBeanDefinitions(
            AnnotationMetadata importingClassMetadata, BeanDefinitionRegistry registry) {
        //向容器中注入AnnotationAwareAspectJAutoProxyCreator这个类
        AopConfigUtils.registerAspectJAnnotationAutoProxyCreatorIfNecessary(registry);

        AnnotationAttributes enableAspectJAutoProxy =
                AnnotationConfigUtils.attributesFor(importingClassMetadata,
EnableAspectJAutoProxy.class);
        if (enableAspectJAutoProxy.getBoolean("proxyTargetClass")) {
            AopConfigUtils.forceAutoProxyCreatorToUseClassProxying(registry);
        }
        if (enableAspectJAutoProxy.getBoolean("exposeProxy")) {
            AopConfigUtils.forceAutoProxyCreatorToExposeProxy(registry);
        }
    }

}
```

我们来分析下这个注入的原理:

1.注册AnnotationAwareAspectJAutoProxyCreator这个bean

断点打在AspectJAutoProxyRegistrar.registerBeanDefinitions()这个函数上 发现调用的堆栈信息还是一个标准的ApplicationContext的调用过程

```
public AnnotationConfigApplicationContext(Class<?>... annotatedClasses) {
    this();
    register(annotatedClasses);
    refresh();
}
refresh() -> invokeBeanFactoryPostProcessors(beanFactory) 果然还是在工厂刚刚创建完毕时调用工厂的后置处理器
```

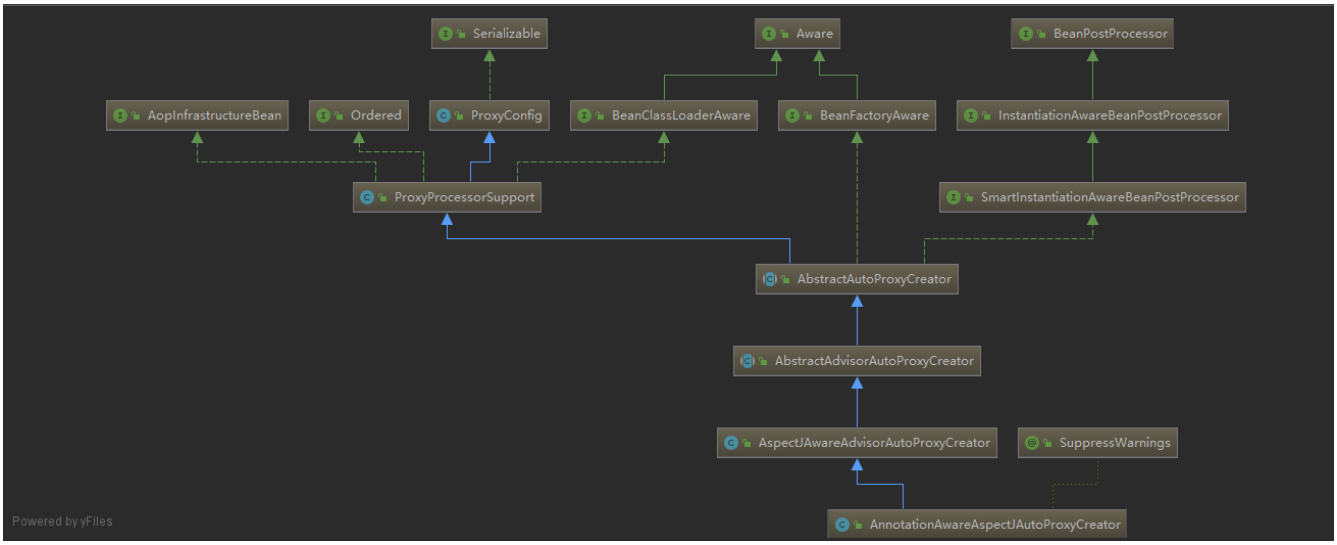
容器中之前就有ConfigurationClassPostProcessor这个类:ConfigurationClassPostProcessor implements BeanDefinitionRegistryPostProcessor implements BeanFactoryPostProcessor ConfigurationClassPostProcessor这个类是BeanFactoryPostProcessor接口的实现类，所以它可以在bean工厂完成后为工厂做功能增强,比如:为bean工厂增加一个新的bean信息，这样就可以在后面的bean创建时直接得到新增bean的实例,而ImportBeanDefinitionRegistrar这个接口就是用来做这个事儿的

```
ConfigurationClassPostProcessor的postProcessBeanDefinitionRegistry
    ->
    processConfigBeanDefinitions(registry);
        ->
        this.reader.loadBeanDefinitions(configClasses);
            ->
            loadBeanDefinitionsForConfigurationClass(configClass, trackedConditionEvaluator);处理配置类信息
                ->

loadBeanDefinitionsFromRegistrars(configClass.getImportBeanDefinitionRegistrars());处理@Import注入类信息
```

```
                    ->
                    entry.getKey().registerBeanDefinitions(entry.getValue(), this.registry);回调
AspectJAutoProxyRegistrar的registerBeanDefinitions()方法
通过一系列的操作,我们发现其实还是通过BeanFactoryPostProcessor这工厂后置处理器进行拦截为我们所想要注入的类进行注入
继续回到AspectJAutoProxyRegistrar类中的registerBeanDefinitions()方法
AopConfigUtils.registerAspectJAnnotationAutoProxyCreatorIfNecessary(registry);
        ->
        registerOrEscalateApcAsRequired(AnnotationAwareAspectJAutoProxyCreator.class, registry,
source);
            ->
            private static BeanDefinition registerOrEscalateApcAsRequired(Class<?> cls,
BeanDefinitionRegistry registry, Object source) {
                Assert.notNull(registry, "BeanDefinitionRegistry must not be null");
                if (registry.containsBeanDefinition(AUTO_PROXY_CREATOR_BEAN_NAME)) {
                    BeanDefinition apcDefinition =
registry.getBeanDefinition(AUTO_PROXY_CREATOR_BEAN_NAME);
                    if (!cls.getName().equals(apcDefinition.getBeanClassName())) {
                        int currentPriority =
findPriorityForClass(apcDefinition.getBeanClassName());
                        int requiredPriority = findPriorityForClass(cls);
                        if (currentPriority < requiredPriority) {
                            apcDefinition.setBeanClassName(cls.getName());
                        }
                    }
                    return null;
                }
                RootBeanDefinition beanDefinition = new RootBeanDefinition(cls);
                beanDefinition.setSource(source);
                beanDefinition.getPropertyValues().add("order", Ordered.HIGHEST_PRECEDENCE);
                beanDefinition.setRole(BeanDefinition.ROLE_INFRASTRUCTURE);
                /*
                为bean工厂中注入AnnotationAwareAspectJAutoProxyCreator
                类名称为
AUTO_PROXY_CREATOR_BEAN_NAME(org.springframework.aop.config.internalAutoProxyCreator)
                */
                registry.registerBeanDefinition(AUTO_PROXY_CREATOR_BEAN_NAME, beanDefinition);
                return beanDefinition;
            }
最终注入了我们想要的类AnnotationAwareAspectJAutoProxyCreator这个类
```

至此我们容器中又被注入了一个名为AnnotationAwareAspectJAutoProxyCreator的新的bean的信息:org.springframework.aop.config.internalAutoProxyCreator=AnnotationAwareAspectJAutoProxyCreator 先看AnnotationAwareAspectJAutoProxyCreator的类的关系图:



AnnotationAwareAspectJAutoProxyCreator ->AspectJAwareAdvisorAutoProxyCreator ->AbstractAdvisorAutoProxyCreator ->AbstractAutoProxyCreator implements SmartInstantiationAwareBeanPostProcessor, BeanFactoryAware 注意:SmartInstantiationAwareBeanPostProcessor extends InstantiationAwareBeanPostProcessor extends BeanPostProcessor 也就是说 AnnotationAwareAspectJAutoProxyCreator这个类实际上是个后置处理器 关注后置处理器（在bean初始化完成前后做事情）、自动装配BeanFactory

关注点1: AbstractAutoProxyCreator.setBeanFactory() AbstractAdvisorAutoProxyCreator.setBeanFactory()->initBeanFactory() AnnotationAwareAspectJAutoProxyCreator.initBeanFactory() 关注点2: AbstractAutoProxyCreator的后置处理器逻辑方法； postProcessAfterInitialization() postProcessBeforeInitialization() postProcessBeforeInstantiation() postProcessAfterInstantiation()

2.注册后置处理器registerBeanPostProcessors(beanFactory); 注册过程在FileSystemXmlApplicationContext源码分析中已经做了说明在此省略但是注意一点: 在注入AnnotationAwareAspectJAutoProxyCreator时会回调我们上面说的关注点1 AbstractAutoProxyCreator.setBeanFactory() AbstractAdvisorAutoProxyCreator.setBeanFactory()->initBeanFactory() AnnotationAwareAspectJAutoProxyCreator.initBeanFactory() 最终创建一个aspectJAdvisorsBuilder

=========以上是创建和注册AnnotationAwareAspectJAutoProxyCreator的过程，此时容器中已经有了这个后置处理器了=========

3.完成最后的单实例的创建finishBeanFactoryInitialization(beanFactory)

1）、遍历获取容器中所有的Bean，依次创建对象getBean(beanName);

getBean->doGetBean()->getSingleton()

2）、创建bean 【AnnotationAwareAspectJAutoProxyCreator在所有bean创建之前会有一个拦截，InstantiationAwareBeanPostProcessor，会调用postProcessBeforeInstantiation()】

```
1）、先从缓存中获取当前bean，如果能获取到，说明bean是之前被创建过的，直接使用，否则再创建；只要创建好的Bean都会被缓存起来
2）、createBean（）；创建bean；
        AnnotationAwareAspectJAutoProxyCreator 会在任何bean创建之前先尝试返回bean的实例
        【BeanPostProcessor是在Bean对象创建完成初始化前后调用的】
        【InstantiationAwareBeanPostProcessor是在创建Bean实例之前先尝试用后置处理器返回对象的】（这里就是整个aop重点的地方，着重分析）
        1）、resolveBeforeInstantiation(beanName, mbdToUse);解析BeforeInstantiation
    希望后置处理器在此能返回一个代理对象；如果能返回代理对象就使用，如果不能就继续
    1）、后置处理器先尝试返回对象；
```

```
    bean = applyBeanPostProcessorsBeforeInstantiation () :
    拿到所有后置处理器，如果是InstantiationAwareBeanPostProcessor;
    就执行postProcessBeforeInstantiation
    if (bean != null) {
        bean = applyBeanPostProcessorsAfterInitialization(bean, beanName);
    }
        2）、doCreateBean(beanName, mbdToUse, args);真正的去创建一个bean实例；流程参考XmlBeanFactory源码分析(下);
3)、AnnotationAwareAspectJAutoProxyCreator【InstantiationAwareBeanPostProcessor】在整个aop中的作用:
        1）、每一个bean创建之前，调用postProcessBeforeInstantiation();
                关心MathCalculator和LogAspect的创建
                1）、判断当前bean是否在advisedBeans中（保存了所有需要增强bean）
                2）、判断当前bean是否是基础类型的Advice、Pointcut、Advisor、AopInfrastructureBean,
                        或者是否是切面（@Aspect）
                3）、是否需要跳过
                        1）、获取候选的增强器（切面里面的通知方法）【List<Advisor> candidateAdvisors】
                                每一个封装的通知方法的增强器是 InstantiationModelAwarePointcutAdvisor;
                                判断每一个增强器是否是 AspectJPointcutAdvisor 类型的；返回true
                        2）、永远返回false

         2）、创建对象
                  调用postProcessAfterInitialization;
                                return wrapIfNecessary(bean, beanName, cacheKey);//包装如果需要的情况下
                                1）、获取当前bean的所有增强器（通知方法）  Object[]  specificInterceptors
                                        1、找到候选的所有的增强器（找哪些通知方法是需要切入当前bean方法的）
                                        2、获取到能在bean使用的增强器。
                                        3、给增强器排序
                                2）、保存当前bean在advisedBeans中;
                                3）、如果当前bean需要增强，创建当前bean的代理对象;
                                        1）、获取所有增强器（通知方法）
                                        2）、保存到proxyFactory
                                        3）、创建代理对象：Spring自动决定
                                                JdkDynamicAopProxy(config);jdk动态代理;
                                                ObjenesisCglibAopProxy(config);cglib的动态代理;
                                4）、给容器中返回当前组件使用cglib增强了的代理对象;
                                5）、以后容器中获取到的就是这个组件的代理对象，执行目标方法的时候，代理对象就会执行通知方法的流程;
```

代码分析:

之前我们在说bean实例的创建时会尝试先返回一个代理对象，整个aop就是这里是关键

```
// Give BeanPostProcessors a chance to return a proxy instead of the target bean instance.
Object bean = resolveBeforeInstantiation(beanName, mbdToUse);//从这里开始
//尝试返回一个代理对象
//以后容器中获取到的就是这个组件的代理对象，执行目标方法的时候，代理对象就会执行通知方法的流程
protected Object resolveBeforeInstantiation(String beanName, RootBeanDefinition mbd) {
        Object bean = null;
        if (!Boolean.FALSE.equals(mbd.beforeInstantiationResolved)) {
                // Make sure bean class is actually resolved at this point.
                if (!mbd.isSynthetic() && hasInstantiationAwareBeanPostProcessors()) {
                        Class<?> targetType = determineTargetType(beanName, mbd);
                        if (targetType != null) {
                /*
                拿到所有后置处理器，如果是InstantiationAwareBeanPostProcessor;
                                                就执行postProcessBeforeInstantiation
```

```
                */
                                bean = applyBeanPostProcessorsBeforeInstantiation(targetType,
beanName);
                                if (bean != null) {
                    //执行postProcessAfterInstantiation
                                        bean = applyBeanPostProcessorsAfterInitialization(bean,
beanName);
                                }
                        }
                }
                mbd.beforeInstantiationResolved = (bean != null);
        }
    //如果返回这个对象不为null那么以后容器就是使用这个代理对象
        return bean;
}
//执行每一个InstantiationAwareBeanPostProcessor类型的后置处理器
protected Object applyBeanPostProcessorsBeforeInstantiation(Class<?> beanClass, String beanName) {
        for (BeanPostProcessor bp : getBeanPostProcessors()) {
                if (bp instanceof InstantiationAwareBeanPostProcessor) {
                        InstantiationAwareBeanPostProcessor ibp =
(InstantiationAwareBeanPostProcessor) bp;
            //执行postProcessBeforeInstantiation()方法
                        Object result = ibp.postProcessBeforeInstantiation(beanClass, beanName);
                        if (result != null) {
                                return result;
                        }
                }
        }
        return null;
}
//AbstractAutoProxyCreator中的关注点2
/*
关注点2:
AbstractAutoProxyCreator的后置处理器逻辑方法;
        postProcessAfterInitialization()
        postProcessBeforeInitialization()
        postProcessBeforeInstantiation()
        postProcessAfterInstantiation()
*/

public Object postProcessBeforeInstantiation(Class<?> beanClass, String beanName) throws
BeansException {
        Object cacheKey = getCacheKey(beanClass, beanName);

        if (beanName == null || !this.targetSourcedBeans.contains(beanName)) {
                //判断当前bean是否在advisedBeans中（保存了所有需要增强bean）
        if (this.advisedBeans.containsKey(cacheKey)) {
                        return null;
                }
        //判断当前bean是否是基础类型的Advice、Pointcut、Advisor、AopInfrastructureBean，或者是否是切面
 (@Aspect)
        /*
        是否需要跳过
                1）、获取候选的增强器（切面里面的通知方法）【List<Advisor> candidateAdvisors】
                        每一个封装的通知方法的增强器是 InstantiationModelAwarePointcutAdvisor;
                        判断每一个增强器是否是 AspectJPointcutAdvisor 类型的；返回true
                2）、永远返回false
```

```
        */
                if (isInfrastructureClass(beanClass) || shouldSkip(beanClass, beanName)) {
                        this.advisedBeans.put(cacheKey, Boolean.FALSE);
                        return null;
                }
        }

        // Create proxy here if we have a custom TargetSource.
        // Suppresses unnecessary default instantiation of the target bean:
        // The TargetSource will handle target instances in a custom fashion.
        if (beanName != null) {
                TargetSource targetSource = getCustomTargetSource(beanClass, beanName);
                if (targetSource != null) {
                        this.targetSourcedBeans.add(beanName);
                        Object[] specificInterceptors = getAdvicesAndAdvisorsForBean(beanClass,
beanName, targetSource);
                        Object proxy = createProxy(beanClass, beanName, specificInterceptors,
targetSource);
                        this.proxyTypes.put(cacheKey, proxy.getClass());
                        return proxy;
                }
        }

        return null;
}

public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {
        if (bean != null) {
                Object cacheKey = getCacheKey(bean.getClass(), beanName);
                if (!this.earlyProxyReferences.contains(cacheKey)) {
                        return wrapIfNecessary(bean, beanName, cacheKey);
                }
        }
        return bean;
}
protected Object wrapIfNecessary(Object bean, String beanName, Object cacheKey) {
        if (beanName != null && this.targetSourcedBeans.contains(beanName)) {
                return bean;
        }
        if (Boolean.FALSE.equals(this.advisedBeans.get(cacheKey))) {
                return bean;
        }
        if (isInfrastructureClass(bean.getClass()) || shouldSkip(bean.getClass(), beanName)) {
                this.advisedBeans.put(cacheKey, Boolean.FALSE);
                return bean;
        }

        // Create proxy if we have advice.
    //找到增强器(其实就是我们配置的通知方法)
        Object[] specificInterceptors = getAdvicesAndAdvisorsForBean(bean.getClass(), beanName,
null);
        if (specificInterceptors != DO_NOT_PROXY) {
        //保存当前bean在advisedBeans中
                this.advisedBeans.put(cacheKey, Boolean.TRUE);
        //如果当前bean需要增强，创建当前bean的代理对象
                Object proxy = createProxy(
                                bean.getClass(), beanName, specificInterceptors, new
```

```
SingletonTargetSource(bean));
                this.proxyTypes.put(cacheKey, proxy.getClass());
                return proxy;
        }

        this.advisedBeans.put(cacheKey, Boolean.FALSE);
    //给容器中返回当前组件增强了的代理对象
    //以后容器中获取到的就是这个组件的代理对象，执行目标方法的时候，代理对象就会执行通知方法的流程
        return bean;
}

protected Object[] getAdvicesAndAdvisorsForBean(Class<?> beanClass, String beanName, TargetSource
targetSource) {
        //找到可用的增强器
    List<Advisor> advisors = findEligibleAdvisors(beanClass, beanName);
        if (advisors.isEmpty()) {
                return DO_NOT_PROXY;
        }
        return advisors.toArray();
}
protected List<Advisor> findEligibleAdvisors(Class<?> beanClass, String beanName) {
    //找到候选的所有的增强器（找哪些通知方法是需要切入当前bean方法的）
        List<Advisor> candidateAdvisors = findCandidateAdvisors();
    //获取到能在bean使用的增强器(原理就是根据切入点去匹配)
        List<Advisor> eligibleAdvisors = findAdvisorsThatCanApply(candidateAdvisors, beanClass,
beanName);
        extendAdvisors(eligibleAdvisors);
        if (!eligibleAdvisors.isEmpty()) {
        //给增强器排序 此处很重要因为会直接导致后面的增强器链的执行顺序
        /*
        因为目标方法和aop增强的方法是有顺序的
        @Before -> 目标方法 -> @After -> @AfterReturning(是否有异常？没有执行方法，有就抛@AfterThrowing)
        */
                eligibleAdvisors = sortAdvisors(eligibleAdvisors);
        }
        return eligibleAdvisors;
}

protected Object createProxy(
                        Class<?> beanClass, String beanName, Object[] specificInterceptors,
TargetSource targetSource) {

        if (this.beanFactory instanceof ConfigurableListableBeanFactory) {
                AutoProxyUtils.exposeTargetClass((ConfigurableListableBeanFactory) this.beanFactory,
beanName, beanClass);
        }

        ProxyFactory proxyFactory = new ProxyFactory();
        proxyFactory.copyFrom(this);

        if (!proxyFactory.isProxyTargetClass()) {
                if (shouldProxyTargetClass(beanClass, beanName)) {
                        proxyFactory.setProxyTargetClass(true);
                }
                else {
                        evaluateProxyInterfaces(beanClass, proxyFactory);
                }
```

```
        }
        //获取所有增强器（通知方法）
        Advisor[] advisors = buildAdvisors(beanName, specificInterceptors);
    //保存到proxyFactory
        proxyFactory.addAdvisors(advisors);
        proxyFactory.setTargetSource(targetSource);
        customizeProxyFactory(proxyFactory);

        proxyFactory.setFrozen(this.freezeProxy);
        if (advisorsPreFiltered()) {
                proxyFactory.setPreFiltered(true);
        }
        //创建代理对象
        return proxyFactory.getProxy(getProxyClassLoader());
}
//创建代理对象：Spring自动决定
public AopProxy createAopProxy(AdvisedSupport config) throws AopConfigException {
        if (config.isOptimize() || config.isProxyTargetClass() ||
hasNoUserSuppliedProxyInterfaces(config)) {
                Class<?> targetClass = config.getTargetClass();
                if (targetClass == null) {
                        throw new AopConfigException("TargetSource cannot determine target class: "
+
                                        "Either an interface or a target is required for proxy
creation.");
                }
                if (targetClass.isInterface() || Proxy.isProxyClass(targetClass)) {
                        return new JdkDynamicAopProxy(config);
                }
        //jdk动态代理
                return new ObjenesisCglibAopProxy(config);
        }
        else {
        //cglib的动态代理
                return new JdkDynamicAopProxy(config);
        }
}
```

4.aop的执行过程

目标方法执行,eg:mathCalculator.div(2,0)

容器中保存了组件的代理对象（cglib增强后的对象），这个对象里面保存了详细信息（比如增强器，目标对象，xxx）

```
1）、CglibAopProxy.intercept();拦截目标方法的执行
2）、根据ProxyFactory对象获取将要执行的目标方法拦截器链；
        List<Object> chain = this.advised.getInterceptorsAndDynamicInterceptionAdvice(method,
targetClass);
                1）、List<Object> interceptorList保存所有拦截器 5
                        一个默认的ExposeInvocationInterceptor 和 4个增强器(本测试用例中是4个切面方法)；
                2）、遍历所有的增强器，将其转为Interceptor；
                        registry.getInterceptors(advisor);
                3）、将增强器转为List<MethodInterceptor>；
                        如果是MethodInterceptor，直接加入到集合中
                        如果不是，使用AdvisorAdapter将增强器转为MethodInterceptor；
                        转换完成返回MethodInterceptor数组；
```

```
        拦截器链（其实就是每一个通知方法又被包装为方法拦截器，后来每一个方法的执行就是利用MethodInterceptor机制）
3）、如果没有拦截器链，直接执行目标方法；

4）、如果有拦截器链，把需要执行的目标对象，目标方法，
        拦截器链等信息传入创建一个 CglibMethodInvocation 对象，
        并调用 Object retVal =  CglibMethodInvocation对象.proceed();
5）、拦截器链的触发过程；
                1)、如果没有拦截器执行执行目标方法，或者拦截器的索引和拦截器数组-1大小一样（执行到了最后一个拦截器）执行目标方法；
                2)、链式获取每一个拦截器，拦截器执行invoke方法，每一个拦截器等待下一个拦截器执行完成返回以后再来执行；
                        拦截器链的机制，保证通知方法与目标方法的执行顺序；
```

```
public Object intercept(Object proxy, Method method, Object[] args, MethodProxy methodProxy) throws Throwable {
        Object oldProxy = null;
        boolean setProxyContext = false;
        Class<?> targetClass = null;
        Object target = null;
        try {
                if (this.advised.exposeProxy) {
                        // Make invocation available if necessary.
                        oldProxy = AopContext.setCurrentProxy(proxy);
                        setProxyContext = true;
                }
                // May be null. Get as late as possible to minimize the time we
                // "own" the target, in case it comes from a pool...
                target = getTarget();
                if (target != null) {
                        targetClass = target.getClass();
                }
        //根据ProxyFactory对象获取将要执行的目标方法拦截器链
                List<Object> chain =
this.advised.getInterceptorsAndDynamicInterceptionAdvice(method, targetClass);
                Object retVal;
                // Check whether we only have one InvokerInterceptor: that is,
                // no real advice, but just reflective invocation of the target.
                if (chain.isEmpty() && Modifier.isPublic(method.getModifiers())) {
                        // We can skip creating a MethodInvocation: just invoke the target directly.
                        // Note that the final invoker must be an InvokerInterceptor, so we know
                        // it does nothing but a reflective operation on the target, and no hot
                        // swapping or fancy proxying.
                        //如果没有拦截器链，直接执行目标方法
            Object[] argsToUse = AopProxyUtils.adaptArgumentsIfNecessary(method, args);
                        retVal = methodProxy.invoke(target, argsToUse);
                }
                else {
                        // We need to create a method invocation...
            /*
            如果有拦截器链，把需要执行的目标对象，目标方法，
                        拦截器链等信息传入创建一个 CglibMethodInvocation对象，
                        并调用 Object retVal = CglibMethodInvocation对象.proceed();
            */
                        retVal = new CglibMethodInvocation(proxy, target, method, args, targetClass,
chain, methodProxy).proceed();
                }
```

```
                retVal = processReturnType(proxy, target, method, retVal);
                return retVal;
        }
        finally {
                if (target != null) {
                        releaseTarget(target);
                }
                if (setProxyContext) {
                        // Restore old proxy.
                        AopContext.setCurrentProxy(oldProxy);
                }
        }
}
public List<Object> getInterceptorsAndDynamicInterceptionAdvice(Method method, Class<?> targetClass)
{
        MethodCacheKey cacheKey = new MethodCacheKey(method);
        List<Object> cached = this.methodCache.get(cacheKey);
        if (cached == null) {
                cached = this.advisorChainFactory.getInterceptorsAndDynamicInterceptionAdvice(
                                this, method, targetClass);
                this.methodCache.put(cacheKey, cached);
        }
        return cached;
}

public List<Object> getInterceptorsAndDynamicInterceptionAdvice(
                Advised config, Method method, Class<?> targetClass) {

        // This is somewhat tricky... We have to process introductions first,
        // but we need to preserve order in the ultimate list.
    //创建一个List<Object> interceptorList保存所有拦截器
        List<Object> interceptorList = new ArrayList<Object>(config.getAdvisors().length);
        Class<?> actualClass = (targetClass != null ? targetClass : method.getDeclaringClass());
        boolean hasIntroductions = hasMatchingIntroductions(config, actualClass);
        AdvisorAdapterRegistry registry = GlobalAdvisorAdapterRegistry.getInstance();
        //遍历所有的增强器 一句话将其转为Interceptor
        for (Advisor advisor : config.getAdvisors()) {
        //如果是切入点的增强器
                if (advisor instanceof PointcutAdvisor) {
                        // Add it conditionally.
                        PointcutAdvisor pointcutAdvisor = (PointcutAdvisor) advisor;
                        if (config.isPreFiltered() ||
pointcutAdvisor.getPointcut().getClassFilter().matches(actualClass)) {
                                //全部转化为MethodInterceptor这种类型的增强器
                MethodInterceptor[] interceptors = registry.getInterceptors(advisor);
                                MethodMatcher mm = pointcutAdvisor.getPointcut().getMethodMatcher();
                                if (MethodMatchers.matches(mm, method, actualClass,
hasIntroductions)) {
                                        if (mm.isRuntime()) {
                                                // Creating a new object instance in the
getInterceptors() method
                                                // isn't a problem as we normally cache created
chains.
                                                for (MethodInterceptor interceptor : interceptors) {
                                                        interceptorList.add(new
InterceptorAndDynamicMethodMatcher(interceptor, mm));
                                                }
```

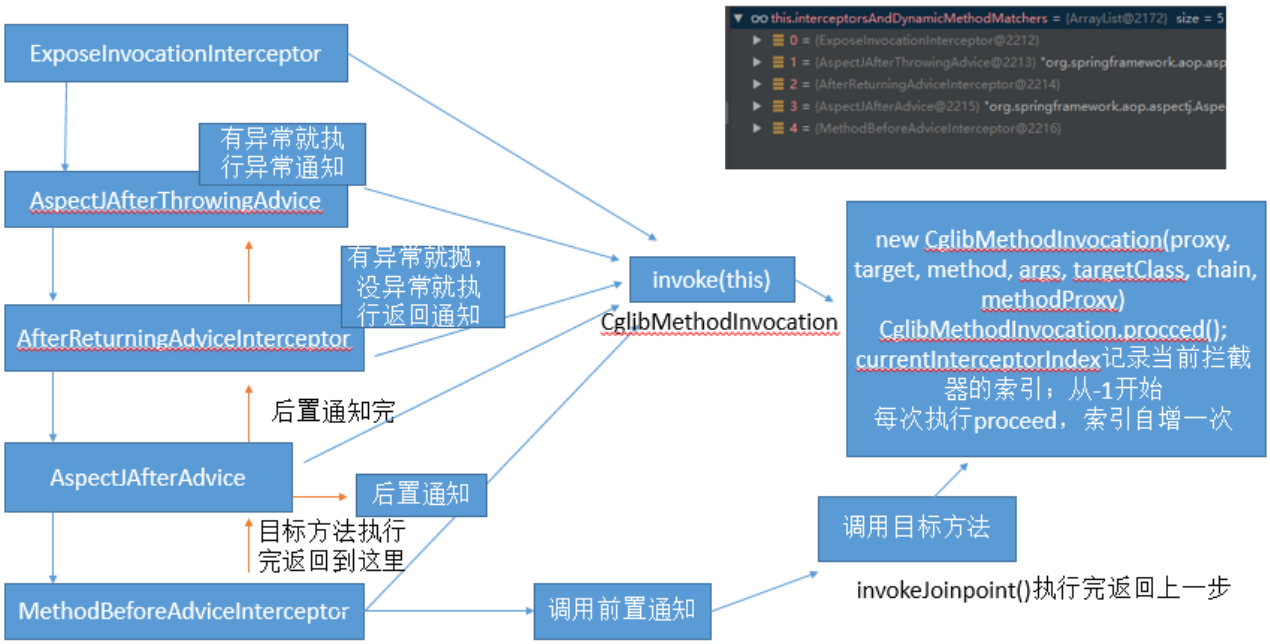
```
                                        }
                                        else {
                                                interceptorList.addAll(Arrays.asList(interceptors));
                                        }
                                }
                        }
                }
                else if (advisor instanceof IntroductionAdvisor) {
                        IntroductionAdvisor ia = (IntroductionAdvisor) advisor;
                        if (config.isPreFiltered() || ia.getClassFilter().matches(actualClass)) {
                                Interceptor[] interceptors = registry.getInterceptors(advisor);
                                interceptorList.addAll(Arrays.asList(interceptors));
                        }
                }
                else {
                        Interceptor[] interceptors = registry.getInterceptors(advisor);
                        interceptorList.addAll(Arrays.asList(interceptors));
                }
        }

        return interceptorList;
}

public MethodInterceptor[] getInterceptors(Advisor advisor) throws UnknownAdviceTypeException {
        List<MethodInterceptor> interceptors = new ArrayList<MethodInterceptor>(3);
        Advice advice = advisor.getAdvice();
        if (advice instanceof MethodInterceptor) {
        //如果是MethodInterceptor，直接加入到集合中
                interceptors.add((MethodInterceptor) advice);
        }
        for (AdvisorAdapter adapter : this.adapters) {
        //如果不是 则需要适配器帮忙
                if (adapter.supportsAdvice(advice)) {
            //使用AdvisorAdapter将增强器转为MethodInterceptor
                        interceptors.add(adapter.getInterceptor(advisor));
                }
        }
        if (interceptors.isEmpty()) {
                throw new UnknownAdviceTypeException(advisor.getAdvice());
        }
    //最后全部变为MethodInterceptor  并返回数组
        return interceptors.toArray(new MethodInterceptor[interceptors.size()]);
}
//拦截器链的触发过程
public Object proceed() throws Throwable {
        //      We start with an index of -1 and increment early.
    //如果没有拦截器执行执行目标方法，或者拦截器的索引和拦截器数组-1大小一样（执行到了最后一个拦截器）执行目标方法
        if (this.currentInterceptorIndex == this.interceptorsAndDynamicMethodMatchers.size() - 1) {
                //调用目标方法
        return invokeJoinpoint();
        }
        //每次得到一个拦截器前先前加加一次
        Object interceptorOrInterceptionAdvice =

this.interceptorsAndDynamicMethodMatchers.get(++this.currentInterceptorIndex);
        if (interceptorOrInterceptionAdvice instanceof InterceptorAndDynamicMethodMatcher) {
                // Evaluate dynamic method matcher here: static part will already have
```

```
                // been evaluated and found to match.
                InterceptorAndDynamicMethodMatcher dm =
                                (InterceptorAndDynamicMethodMatcher)
interceptorOrInterceptionAdvice;
                if (dm.methodMatcher.matches(this.method, this.targetClass, this.arguments)) {
                        return dm.interceptor.invoke(this);
                }
                else {
                        // Dynamic matching failed.
                        // Skip this interceptor and invoke the next in the chain.
                        return proceed();
                }
        }
        else {
                // It's an interceptor, so we just invoke it: The pointcut will have
                // been evaluated statically before this object was constructed.
        //拦截器执行 执行逻辑较复杂  见下图
                return ((MethodInterceptor) interceptorOrInterceptionAdvice).invoke(this);
        }
}
```



注:拦截器链的执行逻辑图

整个aop的总结: 1）、@EnableAspectJAutoProxy 开启AOP功能 2）、@EnableAspectJAutoProxy 会给容器中注册一个组件 AnnotationAwareAspectJAutoProxyCreator 3）、AnnotationAwareAspectJAutoProxyCreator是一个后置处理器； 4）、容器的创建流程： 1）、registerBeanPostProcessors（）注册后置处理器；创建AnnotationAwareAspectJAutoProxyCreator对象 2）、finishBeanFactoryInitialization（）初始化剩下的单实例bean 1）、创建业务逻辑组件和切面组件 2）、AnnotationAwareAspectJAutoProxyCreator拦截组件的创建过程 3）、组件创建完之后，判断组件是否需要增强 是：切面的通知方法，包装成增强器（Advisor）;给业务逻辑组件创建一个代理对象（cglib）； 5）、执行目标方法： 1）、代理对象执行目标方法 2）、CglibAopProxy.intercept()； 1）、得到目标方法的拦截器链（增强器包装成拦截器MethodInterceptor） 2）、利用拦截器的链式机制，依次进入每一个拦截器进行执行； 3）、效果： 正常执行：前置通知-》目标方法-》后置通知-》返回通知 出现异常：前置通知-》目标方法-》后置通知-》异常通知